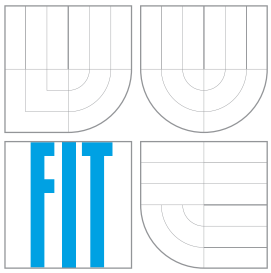
BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ



FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

DIALOG EDITOR IN ANGULARJS FOR MANAGEIQ

EDITOR DIALOGŮ V ANGULARJS PRO MANAGEIQ

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR
AUTOR PRÁCE
ROMAN BLANCO

SUPERVISOR
VEDOUCÍ PRÁCE
Prof. Ing. VOJNAR TOMÁŠ, Ph.D.

BRNO 2016

Abstract

The main goal of this bachelor thesis is to design and implement a new solution for a Dialog Editor for the ManageIQ application. The new editor is supposed to be created as a single-page application implemented by using JavaScript library AngularJS and drag&drop technique. The solution should bring more comfortable interface for end users than the current editor does.

Abstrakt

Hlavním cílem této bakalárské práce je návrh a implementace řešení pro editor dialogů pro aplikaci ManageIQ. Nový editor by měl být vytvořený jako jedno-stránová aplikace vytvořená pomocí JavaScriptové knihovny AngularJS a drag&drop techniky. Řešení mělo poskytnout komfortnější rozhraní koncovým uživatelůn, než nabízí současná implementace.

Keywords

ManageIQ, AngularJS, JavaScript, HTML, user interface.

Klíčová slova

ManageIQ, AngularJS, JavaScript, HTML, uživatelská rozhraní.

Reference

BLANCO, Roman. *Dialog Editor in AngularJS for ManageIQ*. Brno, 2016. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Vojnar Tomáš.

Dialog Editor in AngularJS for ManageIQ

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of prof. Ing. Tomáš Vojnar, Ph.D. and consultant Mgr. Martin Povolný. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Roman Blanco
May 18, 2016

Acknowledgements

I would first like to thank my thesis supervisor prof. Ing. Tomáš Vojnar, Ph.D. and consultant Mgr. Martin Povolný for leading my bachelor thesis. I would also like to thank the members of the Red Hat CloudForms and the Red Hat UX Design teams for their advice and help. Finally I would like to thank my family for all the support throughout my years of study.

Contents

Chapter 1

Introduction

1.1 Motivation

This bachelor thesis contributes to the community-driven project ManageIQ, a tool for cloud and virtualization services.

The main goal is to provide a new implementation for the outdated Dialog Editor in the ManageIQ Admin and Operations User Interface while keeping in touch with modern web 2.0 righ client interfaces. Its current implementation is done in the interpreted programming language Ruby and requires evaluation on the server, that is generally time consuming, because of the need to exchange the data between user and server after almost every user's interaction with the editor, and this forces the user to spend an uncomfortably long amount of time in the editor.

The work on this bachelor thesis is done in collaboration with the Red Hat company, specificaly with the CloudForms and the UX Design team.

The goal was reached by using the AngularJS JavaScript framework and several light-weight JavaScript libraries that will be described later in the document and the Patternfly UI framework, developed by Red Hat UX Design team.

1.2 Outline

In this thesis there will be an analysis of the implementation and familiarization with every part related to the implementation itself. The ManageIQ project will be described at the beginning of this document in Chapter 2. In Chapter 3 there will be an analysis of the current status of the Dialog Editor. In Chapter 4 I will describe how I designed the new version of the Dialog Editor. Used technologies for the implementation will be described in Chapter 5. Explanation of the new Dialog Editor implementation will be in the Chapter 6 and its effectivity and the users comfort will be presented in Chapter 7. The results of the work will be summarized in Chapter 8.

Chapter 2

ManageIQ

ManageIQ is a management and automation platform that gives administrators the possibility to simplify control of their virtual, private, and hybrid cloud infrastructures. Many cloud based projects, like Openstack, Amazon EC2, Google Compute Engine or Microsoft Azure are supported by ManageIQ and that allows administrators to manage the diverse environments by one tool. It can automatically discover these environments wherever they are running and bring them all under one management roof. [3]

ManageIQ is an open source community-driven project implemented in Ruby on Rails. It is developed since 2006, when ManageIQ, Inc. was founded as a startup. In late 2012 ManageIQ was acquired by Red Hat to be used as the basis for the Red Hat CloudForms hybrid cloud management platform, and has been open-sourced in June 2014 [7]. At the time of open-sourcing ManageIQ, the dual-license GPL and Apache License had to be used, but all the time the intention was to be single-licensed under the Apache License exclusively. This step was finaly achieved in March 2016 and was announced with an explanation of the choice on the ManageIQ project site [1] [1].

All the future plans of the ManageIQ community are described in the collaborative organization tool Trello [2] and can be discussed in many ways, mostly through IRC or discussion forums.

A part of ManageIQ is a standalone interface called the Self Service User Interface that was introduced in Red Hat CloudForms 4.0 in December 2015 [3]. It is a portal allowing end users to manage their own services in a simpler interface than ManageIQ itself [2].

In the ManageIQ Admin and Operations User Interface the user can specify a **service dialog** that is used at the moment of the ordering of a service. Also another possible usage of the service dialog is for service automation. For such dialogs we need to have a Dialog Editor in ManageIQ Self Service User Interface, comfortable and easy to use for end users.

Development of the ManageIQ Self Service User Interface is taking place in its own repository, because the whole interface is a 100% REST API driven JavaScript application.

The source code for the ManageIQ Self Service User Interface can be found on the web-based Git repository hosting service GitHub [4].

The reason for mentioning the Self Service User Interface in this thesis is that this standalone interface is where the new Dialog Editor will be used.

[1] https://manageiq.org/
[2] https://trello.com
[3] http://cloudformsblog.redhat.com/2015/11/08/red-hat-cloudforms-4-0-public-beta-2/
[4] https://github.com/ManageIQ/manageiq-ui-self_service

Chapter 3

Analysis of the current status

The initial step to implementation was an investigation of the current implementation. There was a need to find answers for some major questions related to the implementation, such as: *"What is the Service Dialog?"*, *"What does the Service Dialog consist of?"* or *"What are the required preferences for the Dialog?"*.

3.1 Service Dialog

In the ManageIQ interface, Service Dialogs can be found under the *Automate* → *Customization* tab, and there under accordion with the label *Service Dialogs*. Service Dialogs can be used in several situations. The most common, when user wants to provision [1] a service. The dialog in its final form is basically an HTML form that is used to take input from the user. This input is then passed to Automation methods.

Creating a new dialog can be divided into eight basic steps:

1. Navigate to the previously mentioned tab *Automate* → *Customization*.
2. Select the accordion with the label *Service Dialog*.
3. From the toolbar select the *Configuration* → *Add a new Dialog* option.
4. Fill in the information for the Dialog.
5. Fill in the information for a Tab (or more Tabs).
6. Fill in the information for a Box (or more Boxes).
7. Add and configure Elements in the Boxes.
8. Save the dialog.

Every dialog has at least one tab that has been assigned a *Description* and a *Label*. The user can also choose whether the form should have a *Submit* and *Cancel* button, as you can see in Figure 3.1.

After adding and filling in all the required information for a new tab, the user can add a box that will be included under the tab. The same information needs to be provided for the box.

[1] obtain cloud service ordered from a cloud service catalog

A part of ManageIQ database scheme that describes tables where the dialog content is stored can be seen in Figure 3.2.

After these two steps the user should be finally able to add dialog elements, as you can see in Figure 3.3.

For dialogs the user has an option to choose from seven different kinds of elements described below:

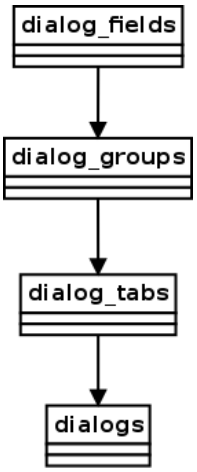


Figure 3.2: Scheme of Dialog

Text box Serves as an input box for text from the user. The user's ability to decide whether the text should be obscured or not is most important for this element. This is useful for entering passwords or any other sensitive data.

Text area Used for long texts. Compared with a Text box, a Text area does not have the possibility to obscure the user's input.

Check box Use of a checkbox is more or less the same as of the standard HTML check boxes. It gives the possibility to select between two values (checked or unchecked).

Drop down list Compared to the Check box, a Drop down list offers more options to choose from. The options are presented in a drop down list.

Radio button Gives exactly the same possibilities as Drop down list, but options are displayed using HTML radio buttons. The user has the option to choose between a Radio button or a Drop down list, because in some cases displaying all the possible values as options in Radio buttons may not be the most clear solution.

Date control For every situation where the user wants to input a date, a Date control is the most suitable option. The result is an input box where the user can select the required date from a calendar.

Date time control Offers one more option compared to Date control. It allows you to specify the exact time of the selected day. For every dialog it is possible to use only one Date control or Date time control element.

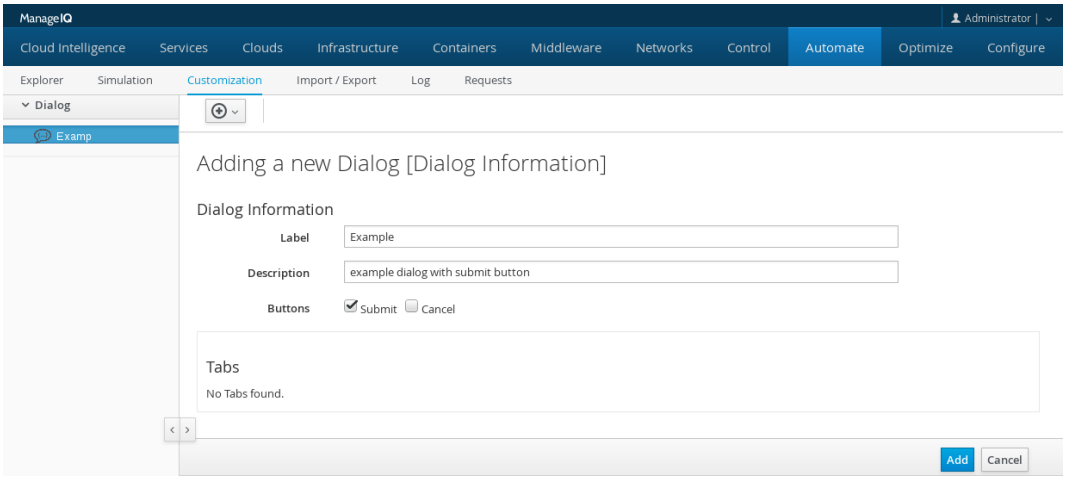


Figure 3.1: Adding new dialog

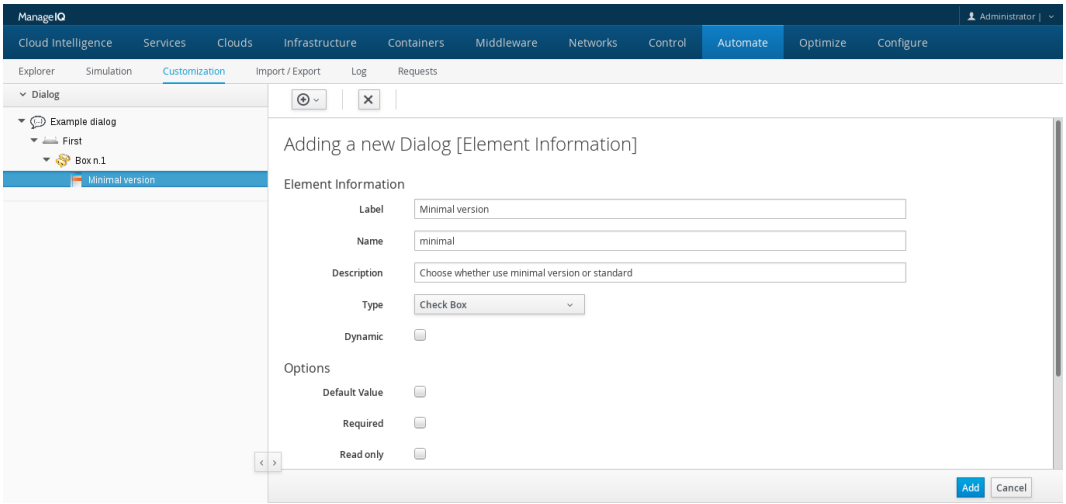


Figure 3.3: Adding new elements to a new dialog

Chapter 4

Design

The main goal of the design was to create a single-page application for the Dialog Editor to solve the problem of loading many pages for various elements of the dialog.

So far the editor included many view templates for the individual steps in the workflow of creating a dialog. That means it was necessary to load a template for each individual steps leading to a high volume for data communication with the server. For that reason, the Dialog Editor was designed as a single-page application from the beginning.

The major requirement for the implementation was to simplify the current solution. For that it was necessary to use the currently used dialog elements described in Section 3.1.

Because in the process of creating the dialog, the users spend the most of their time working with the dialog elements, it was decided that the drag&drop technique would be used for these components. For that reason the toolbox containing individual elements that can be easily dragged to the field representing the dialog content was designed.

In the initial design the toolbox was fixed to the bottom part of the page, so it would be always visible, regardless of the size of the area filled by the created dialog. The only scrollable part would be the content of the dialog. On the left side of the Figure 4.1 there is an example of initialized page with empty dialog. The right part of Figure 4.1 shows filled dialog with scrollable content.

This initial solution was discussed with the UX Design team, that has accepted the idea of using the toolbox, but suggested to split the page into two vertical parts and place the toolbox to the right part of the page, as can be seen in Figure 4.2.

Another option, that is still considered to be used, is to move the elements used for creating the new tab and the new box to the toolbox in the right part of the page. This can be seen in Figure 4.3. The reason for the change is to get together all the elements that are used for creating the dialog. Because it is still not clear if the change will contribute to the user's comfort, this version is still being discussed with the UX Design team.

For the Dialog Editor implementation, the version of the design on Figure 4.2 has been used for the Dialog Editor implementation so far.

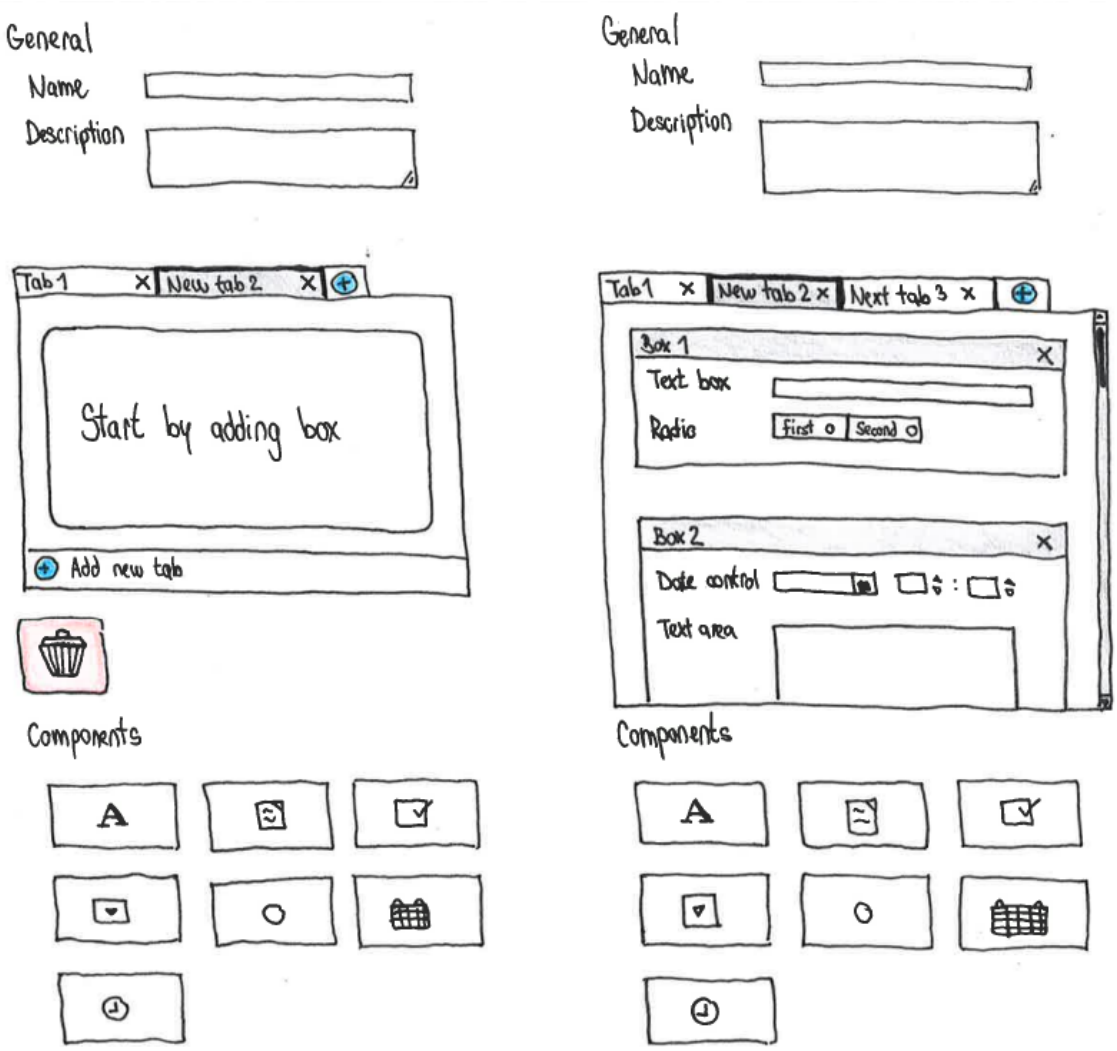


Figure 4.1: Mockup of a first version for implementation

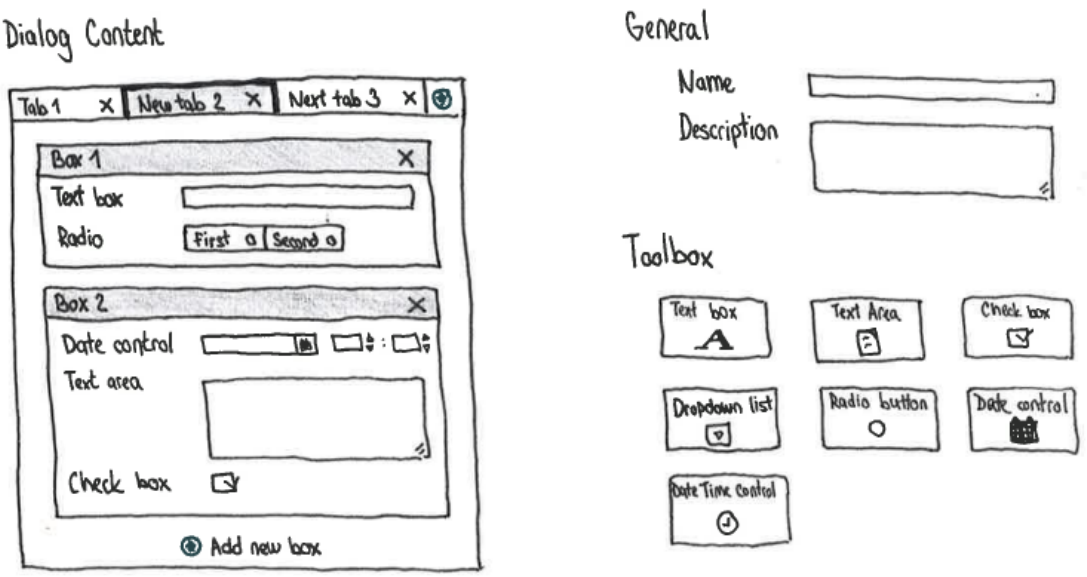


Figure 4.2: Mockup of a second version with the toolbox on the right side of the page

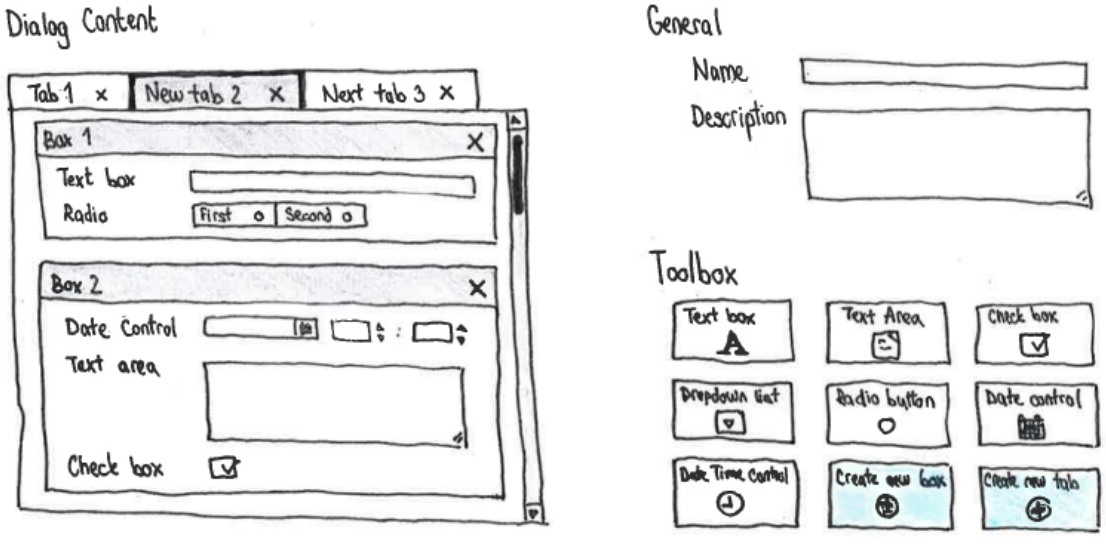


Figure 4.3: Mockup of a third version with the buttons moved into the toolbox

Chapter 5

Used technologies

One of the requirements for the implementation was to use the JavaScript framework AngularJS. On top of that, to make the implementation easier, a few extra JavaScript libraries were used in the project.

5.1 AngularJS

AngularJS [1] is an advanced front-end JavaScript framework released by Google. It offers a way to build rich front-end experience with cutting-edge tools quickly and easily [5].

The decision to use AngularJS library was made by the ManageIQ community. The plan is to reimplement many parts of the project's user interface in AngularJS. Using AngularJS seemed to be an ideal choice for many reasons. It provides an easier way to create modularized code and allows to write structured *DRY*[2] code. Another reason to use AngularJS is the reaction to a user's interaction with the editor. The interaction is evaluated completely in the user's browser, so a server is saved from a lot of unnecessary communication with the client — template loading and reevaluation of state after every change on the jpage.

5.1.1 AngularJS version

The latest stable AngularJS version — was choosend for the implementation, even though at the beginning of the work on the Dialog Editor the version 1.4 was used. The reason for this decision was based on an effort of the ManageIQ project to hold onto the *bleeding edge technology*[3]. To make it possibile to implement the editor in version 1.5, it was, among other things, also necessary to solve the dependencies for the new version. That meant submitting Pull Requests to the different projects' upstreams that ManageIQ uses.

[1] https://angularjs.org/
[2] Don't repeat yourself
[3] the most advanced stage of a technology with risk of being unreliable

5.1.2 Transition to AngularJS 2.0

It is worth mentioning that AngularJS version 2.0 [4] which, at the time of writing this thesis is still in *beta* phase, will not be backwards compatible with previous versions of AngularJS.

AngularJS developers, however, are making an effort to make the transition to the new version 2.0 as smooth as possible by backporting version 2.0 functionalities to the minor releases of AngularJS 1. A nice example is the method `component` that was introduced in version 1.5, as an alternative to the `directive` method used in previous versions. The `component` method allows the creating of new HTML elements with their own JavaScript code — *controller* and HTML template. The same can be achieved with the `directive` method, but only by using a soon to be deprecated style.

A lot of guidance for the transition from AngularJS 1 to AngularJS 2, and even between AngularJS 1 minor versions, can be found on the internet [6], making the situation less problematic. In the following paragraph is well described the reason to keep up with the latest AngularJS versions:

> There are many changes happening, both inside and outside of the "Angular world". The best way to be able to transition to Angular 2.0 is to keep our apps updated as new Angular versions are released. Staying on 1.2 or 1.3 until 2.0 is released is probably going to result in a huge undertaking [4].

5.2 External libraries

5.2.1 angular-dragdrop

As the name of the library suggests, this is the key extension that gives the implemented editor the ability to work with elements by using the drag&drop technique. `angular-dragdrop`[5] is a JavaScript library created as an AngularJS wrapper for the jQueryUI drag&drop functionalities. For that reason, some might notice some familiar attributes in the implemented components that are coming from jQueryUI and will be described in the following part of this document.

The reasons for choosing this library for the implementation is its prevalence, that implies a wider support and development. A further big advantage of this choice is that another planned part of the Self Service User Interface with drag&drop support should use this exact library, however this was not the main reason for the choice, but rather a coincidence.

5.2.2 ui-sortable

Another important library for the implementation of Dialog Editor that also happens to be a wrapper on jQueryUI is `ui-sortable` that is developed as one of a suite of AngularJS directives grouped in a project called *AngularUI*[6].

This library allows a much more efficient way to implement the sorting of elements in dialogs, than would be possible by using just `angular-dragdrop`.

[4] https://angular.io/
[5] https://github.com/codef0rmer/angular-dragdrop
[6] https://angular-ui.github.io/

5.2.3 angular-bootstrap-switch

Not as much important as far as the implementation is concerned, the AngularJS library `angular-bootstrap-switch` is however significant to the appearance of the project.

It gives the option of using *Bootstrap Switches*[7] with the AngularJS library. Bootstrap Switches are basically HTML `input` elements implemented using the, among front-end developers, well known library *Bootstrap*[8]. The resulting switch gives a pleasant effect, and is also easier to use compared to a standard HTML check box, thanks to its size.

[7] http://www.bootstrap-switch.org/
[8] https://getbootstrap.com/

Chapter 6

Implementation

Before starting with the reimplementation of Dialog Editor, I needed to create a whole UI state for dialogs in Self Service.

Before the new implementation, the dialogs in Self Service User Interface were only used in *Service Catalog*, but there was no place where Dialogs could be listed and certainly not edited.

As a first step, a new field had to be added to navigation menu [1]. That meant adding new AngularJS router [2] state, connecting it with the API and making sure that the data are being loaded, in this case the count of available dialogs. In this step it was only needed to modify the JavaScript files that were already in the project.

The next step, after adding a new field *My Dialogs* to the navigation menu, was to add a new list state [3], and connect it with the menu item intended for dialogs. To enable the possibility to list dialogs, in this step a HTML template for the list of dialogs had also to be added. In addition, filtering and sorting of displayed results is in the dialog list had to be solved. Figure C.1 shows how the list state looks like.

When the list state was ready, the next step was to create a dialog detail state [4]. An example of this can be seen in Figure C.2. In this state the user can see how a created dialog looks, and has displayed the most relevant information, like the date when the dialog was created or the last time it was updated.

After these three steps, everything that was necessary to start work on the Dialog Editor had almost been done, the last state that had to be added was the edit state [5], where the Dialog Editor is placed.

6.1 Dialog Editor

6.1.1 Dynamic Tabs

The first part that was created as a part of the dialog editor was Dynamic Tabs [6], because, as was mentioned previously in Subsection 3.1, tabs are on the top level of the data structure for the dialog.

[1] https://github.com/ManageIQ/manageiq-ui-self_service/pull/37/commits/e6c9aa1
[2] http://angular-ui.github.io/ui-router/site/#/api/ui.router
[3] https://github.com/ManageIQ/manageiq-ui-self_service/pull/37/commits/9846a73
[4] https://github.com/ManageIQ/manageiq-ui-self_service/pull/37/commits/4efe83e
[5] https://github.com/ManageIQ/manageiq-ui-self_service/pull/37/commits/77297df
[6] https://github.com/ManageIQ/manageiq-ui-self_service/pull/37/commits/4f4fbb6

The dynamism of the tabs lies in the ability to add new tabs or remove existing tabs, and also to change the order of existing tabs.

When creating the Dynamic tabs, a major step was to create the AngularJS component. The AngularJS component is named `dynamicTabs` in the code, meaning that this component can be called in HTML by a new element `<dynamic-tabs></dynamic-tabs>` created by AngularJS.

That would be the basic description of calling the component, but for our case, the dynamic tabs also need to get data with the dialogs tabs. For that, another handy feature of AngularJS called data binding is used.

In the code of the created component an HTML attribute can be specified, through which the data can be passed, in this case named `tabList`.

For a brief demo, the simplified code for the currently described component looks like the following code:

```
angular.module('app.components')
.component('dynamicTabs', {
  bindings: {
    tabList: '='
 }
})();
```

and usage of it in HTML code would look like this:

```
<dynamic-tabs tab-list='dialog_tabs'></dynamic-tabs>
```

where `dialog_tabs` is a JavaScript object containing all the data that is necessary.

As for manipulating tabs, adding a new tab is handled by a function in the component controller named `addTab`. It works with the previously mentioned data object `tabList`. In the current implementation, the behavior that leads to adding a new tab is:

1. set all current tabs as inactive,
2. create a new object that has the attribute indicating activity set to `true`,
3. push the newly created data object to the `tabList` array.

New tab is always appended to the end.

Behaviour for deleting the tab that is defined in component controller by the name `deleteTab` is separated into two steps. Before deleting the tab, it is necessary to check if the deleted tab is active. In case where it is active, the activity needs to be passed to another tab. Because of that there are two cases for deleting an active tab:

- If the deleted tab was last, the activity goes to the previous tab.
- If the deleted tab has any following tab, the activity goes to the next tab in sequence.

In a case where a deleted tab is not active, it is not necessary to handle the situation, and that tab can be removed from the array.

For both adding and deleting tabs the JavaScript utility library *lodash*[7] was used; this allows easier work with arrays or objects in JavaScript, and came in handy even in other parts of the implementation.

[7] https://lodash.com/

6.1.2 Dialog Edit service

Upon completion of Dynamic tabs it was most essential to clarify how to work with data using more than just one component, that was necessary for the implementation of Dialog Editor.

Creating a *service* seemed like an ideal option. To create a *service* appeared an ideal option. This AngularJS feature makes it possible to share the data between all components where it is needed and update them reflecting the change between all components.

This component is called `DialogEdit`[8] in the implementation. There are two main methods in this service:

- method `setData`, which has simple code that takes the data through a parameter, and stores them in service variable `data`:

```
service.setData = function(data) {
  service.data = data;
};
```

- method `getData`, is also a simple method that returns data stored in `data` variable:

```
service.getData = function() {
  return service.data;
};
```

The third method — `updatePosition`, does not have much common with storing data. It is used for updating `position` attribute for objects that can be sorted in Dialog Editor. Usage of this method will be described in more detail in a later section of this document.

6.1.3 Dialog Dashboard

Another AngularJS component created for the Dialog Editor is *Dialog Dashboard*[9]. All the content for each dialog tab is displayed inside this component. Both Boxes and Fields belonging to the Box are rendered from the template of this component by using the built-in AngularJS directive *ngRepeat*, which makes it possible to write one template that will be used for each item inside an iterable collection.

The most important methods are, similar to *Dynamic Tabs*, the method used for adding new boxes — `addBox` and the method for removing them — `removeBox`.

6.1.4 Dialog Field

A component that has a major influence on the editor feeling like *WYSIWYG*[10] is Dialog Field[11]. Thanks to this component, the user can immediately recognize which types of elements are present in Dialog.

The most important part of this component is its template that includes the HTML code for every element type that can be used in dialog. It consists of one large HTML file that uses *ngSwitch* directive from AngularJS, which renders an HTML element depending on its condition.

[8] https://github.com/ManageIQ/manageiq-ui-self_service/pull/37/commits/69e75a4
[9] https://github.com/ManageIQ/manageiq-ui-self_service/pull/37/commits/69e75a4
[10] What You See Is What You Get
[11] https://github.com/ManageIQ/manageiq-ui-self_service/pull/37/commits/ce782e4

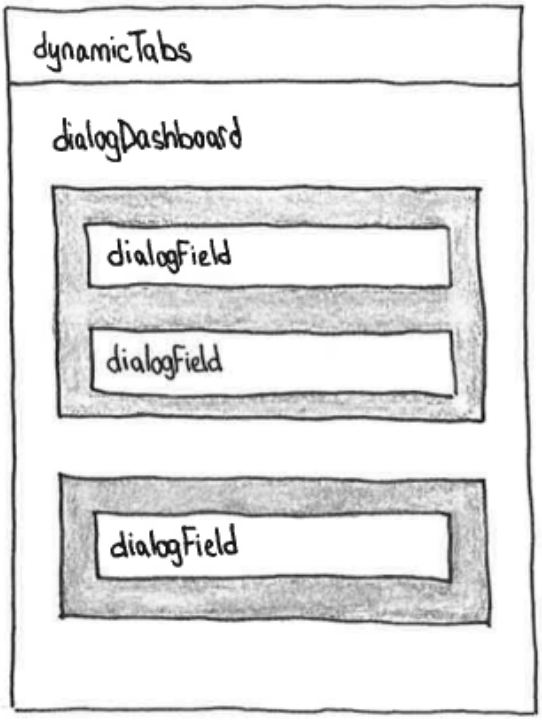


Figure 6.1: Mockup of a part of the Dialog Editor

For a better illustration, the simplified code of the template for this component looks like this:

```
<div ng-switch on="dialogField.fieldData.type">
    <div ng-switch-when="DialogFieldTextBox">
        <!-- HTML code for Text box field -->
    </div>
    <div ng-switch-when="DialogFieldTextAreaBox"> ... </div>
    <div ng-switch-when="DialogFieldCheckBox"> ... </div>
    <!-- etc. -->
</div>
```

In Figure 6.1, it can be seen how the previously described components are integrated together and in Figure C.3 there is an example of how the created dialog looks in reality.

6.1.5 Dialog Edit modal

An important component for editing the details for individual Fields is *Dialog Edit modal*[12]. For every element type, a template through which the user can access and modify the attributes of dialog field is created.

The *Dynamic* is ratger specific. It replaces the original attributes of the element with attributes specific for dynamic elements. This was introduced for every element in ManageIQ release *Botvinnik*.

The described templates are rendered using the AngularJS function `directive`, as

[12] https://github.com/ManageIQ/manageiq-ui-self_service/pull/37/commits/ce782e4

features of the `component` method do not support a way to render a different template according to a parameter.

Here is a simplified *directive* code, where it is possible to see how the directive decides which template file is supposed to be rendered, according to the parameter that is sent by the attribute `template`:

```
.directive('dialogModalTemplate', function() {
  return {
    templateUrl: function(tElement, tAttrs) {
      return 'app/components/dialog-modal-template/' + tAttrs.template;
    },
    scope: true,
  };
});
```

For example, a Text area is then rendered from HTML using this directive by this element:

```
<dialog-modal-template ng-switch-when="DialogFieldTextAreaBox"
                       template="edit-dialog-modal-text-area-box.html">
```

Figure C.4 is showing how a modal looks for the Radio button field.

6.1.6 Draggable components

The last component that was created [13] for the implementation works more or less as a placeholder for new Fields that can be dragged to the Dialog Box.

In the controller all the necessary attributes for all types of elements are described, so after dragging the element to the box, the user can immediately see, how the element will approximately look like.

6.1.7 Drag&Drop and sorting

To allow the user to manipulate objects of the dialog easier than it was in the previous implementation, a mouse is used in the editor for sorting and dragging elements.

As was mentioned in the chapter describing the *ui-sortable* library, it offers a very efficient way of implementing the sorting of elements.

For the element, where the sorting should be applied, the *ngModel* directive is specified, through which data are provided to the library. The data contains an array that is connected with rendered elements, and after changing the position of the element by dragging it to a different place, the position of the element in the array is changed as well.

The definition of the component may even specify the sorting behaviour that jQueryUI offers.

In the implementation, sorting is used for Tabs, Boxes and Dialog Fields. On the other hand, dragging elements is used only for one component — the draggable components that were described in the previous sub-section. The droppable zone for these elements is the Dialog Dashboard, that also has an object in its component's code, where behaviour is specified, again by using jQueryUI possible interactions.

[13] https://github.com/ManageIQ/manageiq-ui-self_service/pull/37/commits/036eef6

Chapter 7

Effectivity and Users comfort

One of the most important goals of this thesis was to improve the effectivity and the users comfort. The main difference in terms of effectivity can be observed on the amount of the data transfered between the server and client, that is significantly lowered by using the new version of the Dialog Editor.

For a better illustration a new dialog was created while observing data flow using the network diagnostic tool Wireshark.

In both cases the created dialog had the same content. It contained only one element — Check Box.

In Figure 7.1 you can see the size of TCP packets transfered between the client and the server in a process of creating the new dialog by using the older version of Dialog Editor in the ManageIQ Admin and Operations User Interface. The summarized stats are displayed in Table 7.1.

Packets	94
Bytes	154290
Average packet size	1641 bytes

Table 7.1: Summary of the TCP data flow between client and server by using the old version of Dialog Editor

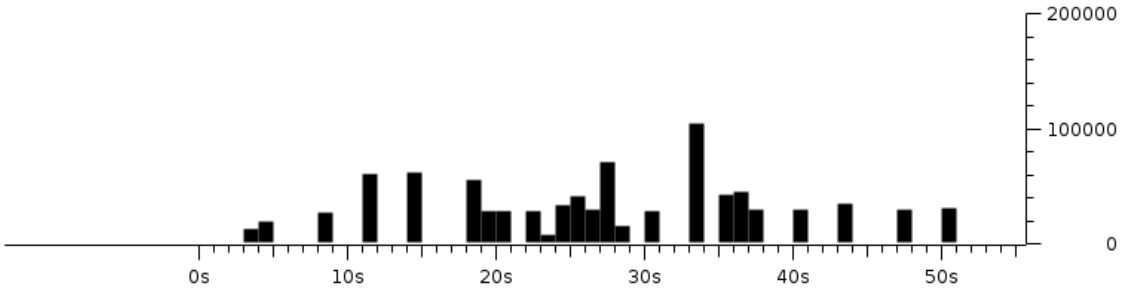


Figure 7.1: Sizes of TCP packets transfered between client and server by using the old version of Dialog Editor

In the case where the new Dialog Editor was used, as can be seen from the Table 7.2, the amount of transfered data was more than 10× lower.

Packets	81
Bytes	13479
Average packet size	166 bytes

Table 7.2: Summary of the TCP data flow between client and server by using the new version of Dialog Editor

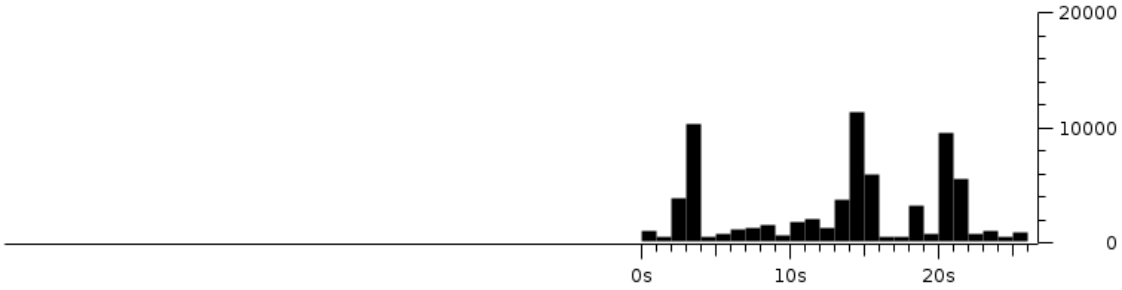


Figure 7.2: Sizes of TCP packets transfered between client and server by using the new version of Dialog Editor

As for memcached, from the graph in Figure 7.3 it can be told, that the amount of data used by memcached is almost the same as the amount of data transfered by the TCP communication between client and server. On the other hand, by using the new Dialog Editor, memcached is only used for storing the session token. As you can see in Figure 7.4, the communication with memcached has occured only once.

Packets	126
Bytes	645948
Average packet size	5127 bytes

Table 7.3: Summary of transmitted memcached data by using the old version of Dialog Editor

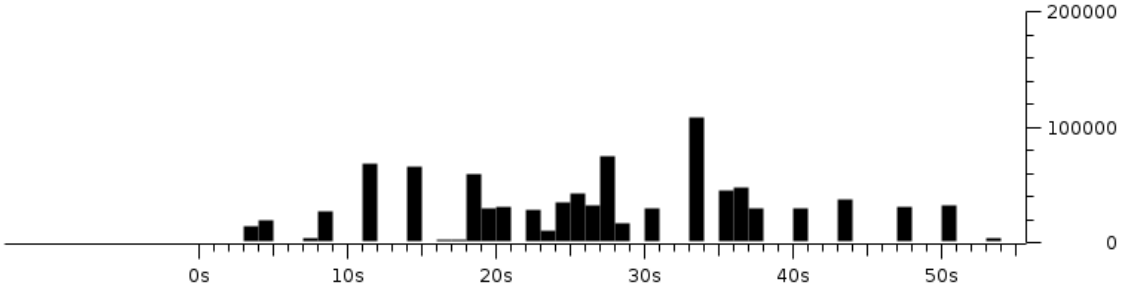


Figure 7.3: Sizes of memcached packets transfered by using the old version of Dialog Editor

Packets	14
Bytes	2295
Average packet size	164 bytes

Table 7.4: Summary of transmitted memcached data by using the new version of Dialog Editor

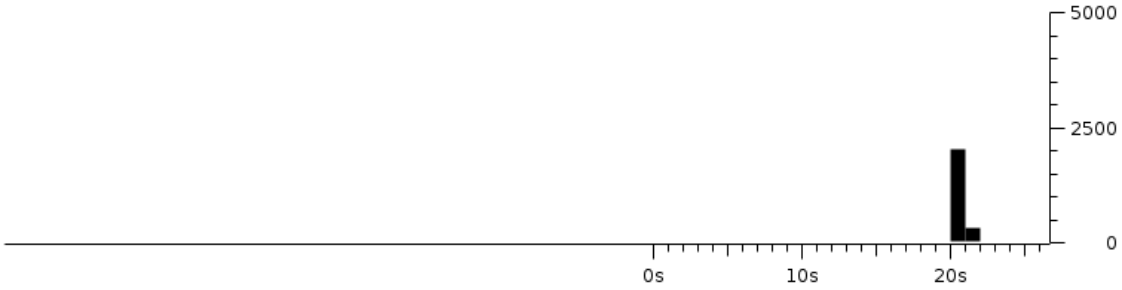


Figure 7.4: Sizes of memcached packets transfered by using the new version of Dialog Editor

As was mentioned at the beginning of the document, the current implementation forces the user to spend an uncomfortably long amount of time in the editor. By giving the user a way to create the dialog by using a drag&drop technique, the workflow of creating a new dialog is much smoother.

As an example we can take the graphs comparing an amount of the data transmitted between the client and the server. Comparing the time axes these, we can see, that the length of the time that was spent by creating the dialog containing one Check box is approximately 2× shorter while using the new version of the dialog editor.

Chapter 8

Conclusion

In my work on the Dialog Editor I have learned to use the AngularJS framework and studied the original version of the Dialog Editor. The gained knowledge allowed me to to implement the new interactive editor for ManageIQ with drag&drop technique together with specs coverage. In this document I've summarized the solution and effectivity of the previous and the new implementation.

One of the biggest advantages of the implementation is that the whole work is created as an open source project that will be used in the future as a separate component, allowing many developers to improve or modify the Dialog Editor for their own purposes.

The result of the work is Dialog Editor that has been introduced as a Pull Request to the project's upstream [1]. At this moment the work is in review status by the members of the Red Hat CloudForms team.

8.1 Ideas for a future development

Even though the goal set by the bachelor thesis was successfuly achieved, a lot of work can still be done in the Dialog Editor to improve the user's experience.

Removing the tabs, boxes and elements is not protected by any confirmation. Especially in the case of tab removal, this could lead to the unwanted situation, when the user would accidentaly remove a part of the dialog. On the other hand, requesting an answer from the user every time he wants to remove a part of the dialog would be uncomfortable from the user's perspective. The better way would be to give the user an option to undo the changes. To achieve this solution, using the `sessionStorage` from the Web Storage API [2] seems to be the best option.

Another improvement could be done in the drag&drop toolbox zone. In the current solution, the element dropped to the dialog's box is always appended at the end of all elements in the box. A more comfortable solution would be to detect where exactly the user has dropped the element, and insert it at the position it was dropped at. Another problem is with sorting elements in the box. Right now it is only possible to sort the elements inside of one box, but not between boxes.

Purely esthetical change, but still related to the drag&drop is the idea to provide a visual suggestion for every draggable element. At the first sight it may not be clear to the users which parts of the Dialog Editor are draggable, and where they can be dropped.

[1] https://github.com/ManageIQ/manageiq-ui-self_service/pull/37

[2] https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API

As was mentioned earlier in Chapter 4, an option to move buttons for creating a new tab and box from the area where the dialog is displayed into the toolbox of draggable elements is also discussed.

The suggestions for the improvements mentioned above will be discussed with the Red Hat UX Design and the CloudForms teams to provide the best possible solution.

Bibliography

[1] Barenboim, O.: *Open Source License Change.*
Retrieved from:
http://manageiq.org/blog/2016/04/open-source-license-change/

[2] Draper, M.: *ManageIQ Capablanca: Azure, Containers, Self-Service UI.*
Retrieved from: http://manageiq.org/blog/2015/12/
manageiq-capablanca-azure-containers-self-service-ui/

[3] Jansen, G.: *Managing heterogeneous environments with ManageIQ.*
Retrieved from: http://lwn.net/Articles/680060/

[4] Johansson, J.-P.: *Preparing for the future of AngularJS.*
Retrieved from: https:
//www.airpair.com/angularjs/posts/preparing-for-the-future-of-angularjs

[5] Lerner, A.: *Riding Rails with AngularJS.* fullstack.io. 2013.

[6] Motto, T.: *Exploring the Angular 1.5 .component() method.*
Retrieved from:
https://toddmotto.com/exploring-the-angular-1-5-component-method/

[7] Red Hat Inc.: *Red Hat Launches First Open Source Release of ManageIQ Software.*
[Online; visited 3.4.2016].
Retrieved from: http://www.redhat.com/en/about/press-releases/
red-hat-launches-first-open-source-release-manageiq-software

Appendices

List of Appendices

Appendix A

Content of the CD

Directories:

- **src** - source code files
  - **latex** - files required for building this document
  - **dialog_editor** - source files for the dialog editor
  - **dialog_editor_vm** - virtual machine set up for launching the Dialog Editor
- **pdf** - PDF version of the thesis

Appendix B

Manual

On the CD attached is a virtual machine image. It is located in the `dialog_editor_vm` directory, ready to be deployed using Virtual Machine Manager.

1. The first step is to create a NAT Virtual Network in Virtual Machine Manager. The manual on *libvirt* project wiki can be used[1].
2. In Virtual Machine Manager select the button with label *Create a new virtual machine*.
3. From options to install the operating system select the one with the label *Import existing disk image*.
4. In the dialog select the virtual machine image from CD and choose memory and CPU settings. (give it at least 2048 MiB of RAM).
5. In the final step, select the network you have created.

After you create a new virtual machine, the machine should start automatically. Upon booting, you can login with default credentials `root` / `smartvm`. To start ManageIQ in your browser you will need an IP address of the machine. The IP address can be found by running this command:

```
$ ip a
```

The ManageIQ Self Service UI can be accessed at `http://[IP ADDRESS]/self_service`.

[1] http://wiki.libvirt.org/page/TaskNATSetupVirtManager

Appendix C

Figures

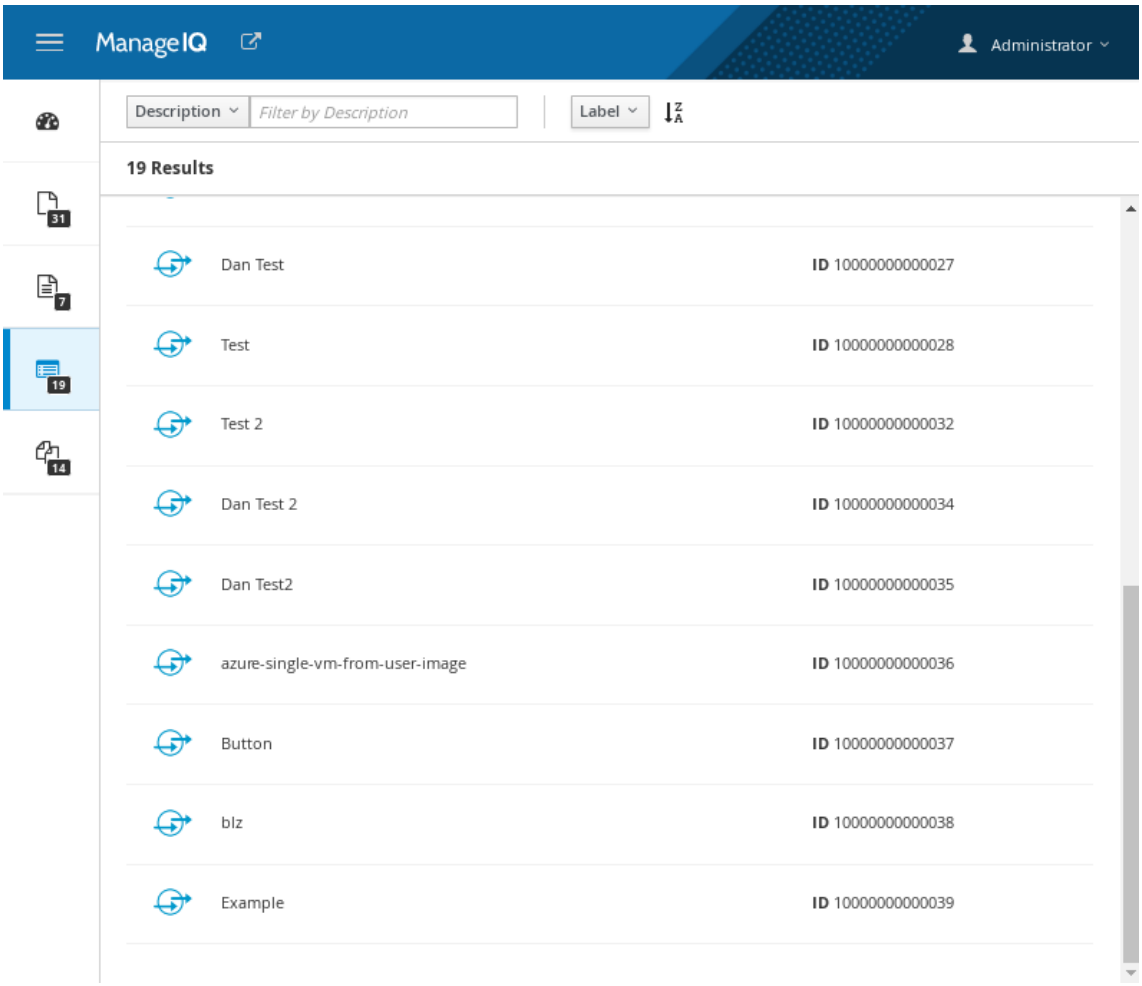


Figure C.1: List state for Dialogs in Self Service User Interface

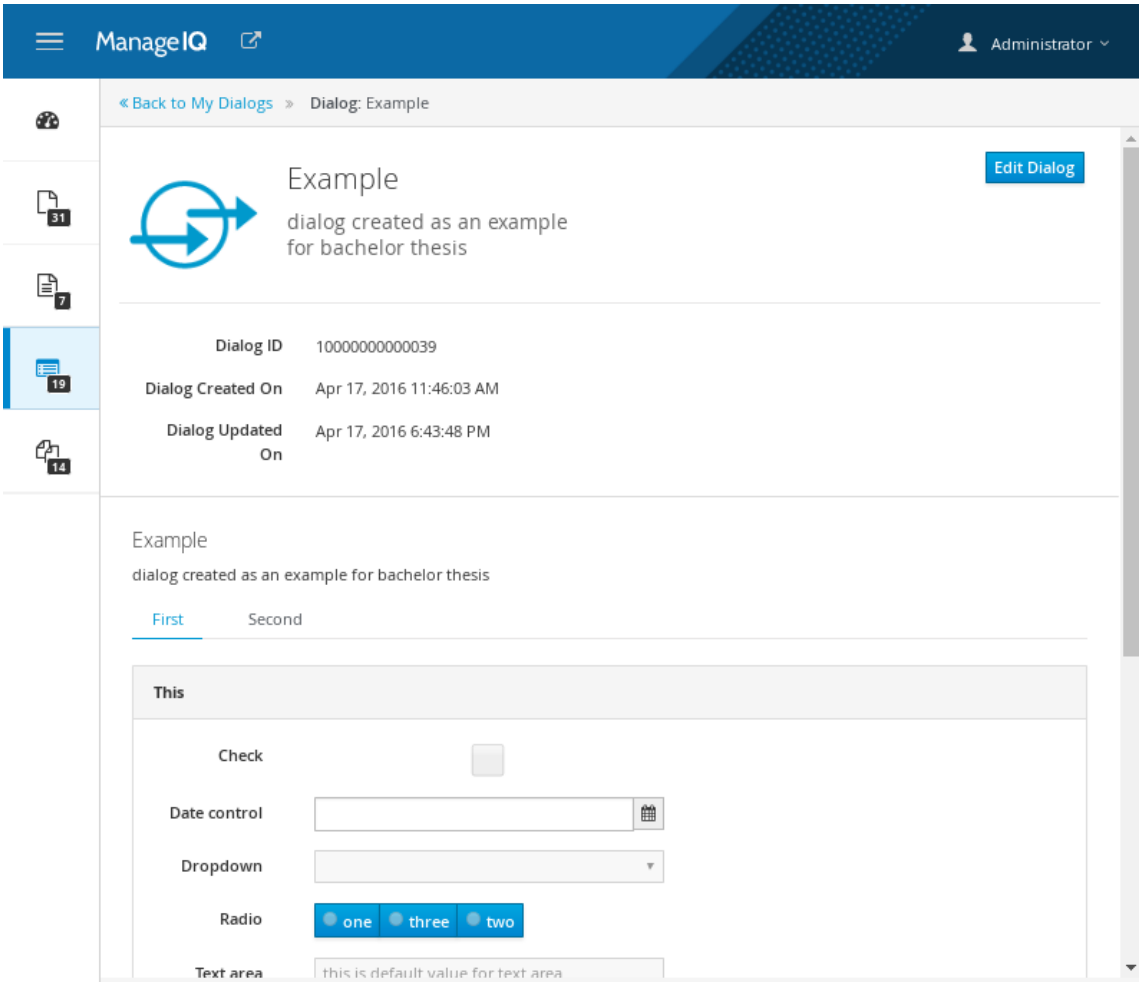


Figure C.2: Detail state for Dialogs in Self Service User Interface

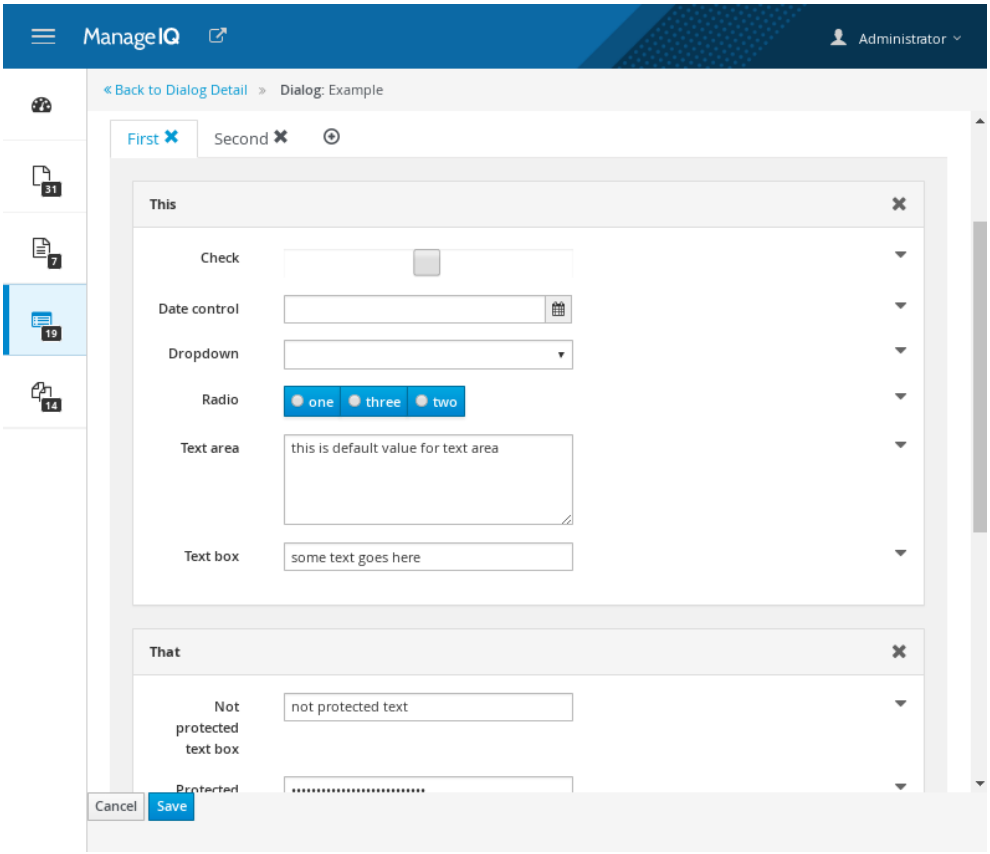


Figure C.3: Example of Dialog in new Dialog Editor

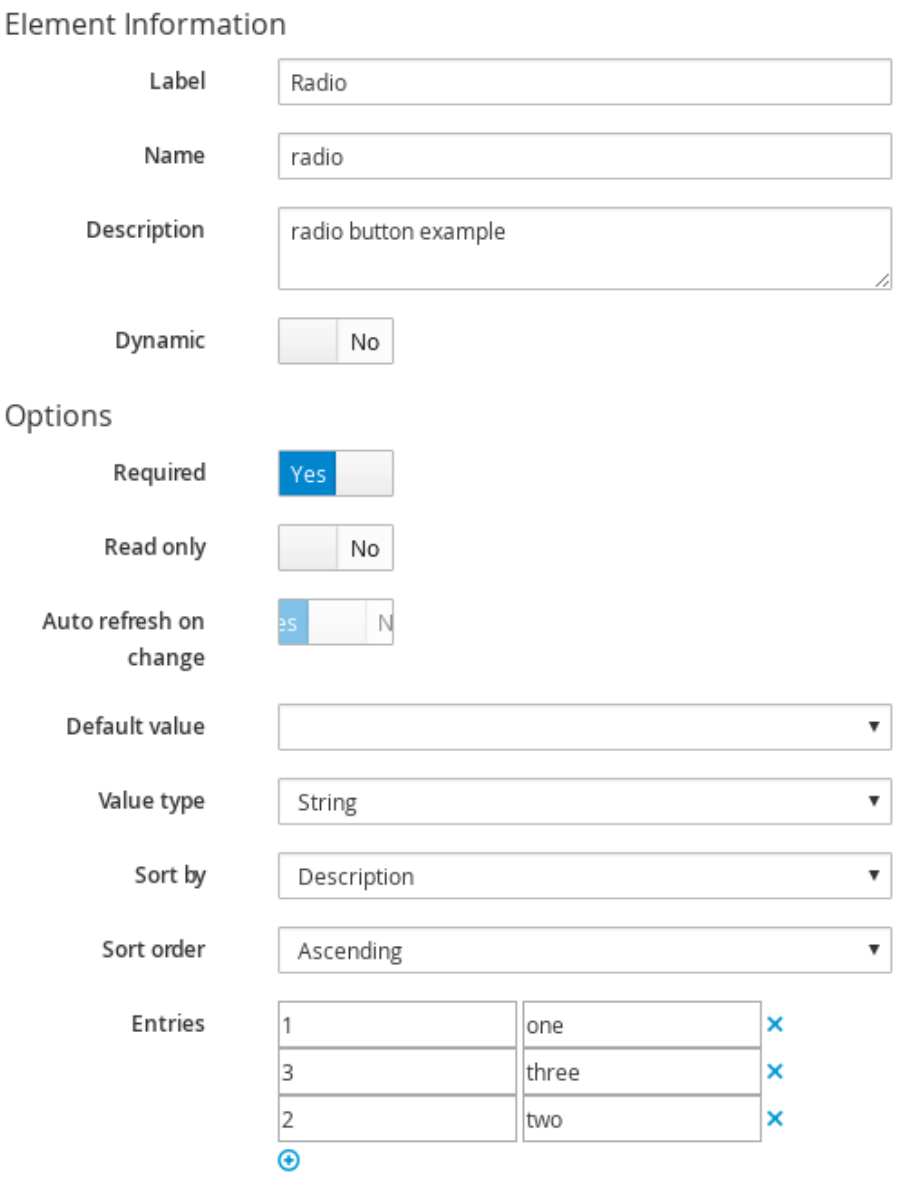


Figure C.4: Modal for the Radio button field